Coupled context free grammars and a programming language based on them

AUTHOR(S):

YAMASHITA, Yoshiyuki; NAKATA, Ikuo

数理解析研究所講究録
618巻 1987年 130-149

Coupled context free grammars and a programming language based on them

Yoshiyuki YAMASHITA and Ikuo NAKATA,
山下義行　　中田育男

Institute of Information Sciences and Electronics,
電子情報工学系

University of Tsukuba,
筑波大学

Abstract

A formal grammar: Coupled Context Free Grammar(CCFG) and its interpretation as a programming language are introduced. Context free grammars(CFGs) can be regarded as the representations of data structures. CFGs can be coupled by coupling their production rules into n-tuples of rules in a CCFG. These couplings can be regarded as the representation of the relation between the data structures expressed by component CFGs. Therefore a CCFG can be regarded as a program. The denotational semantics of the program is defined as a set of n-tuples of terminal strings derived by the grammar. It is easily understood that logic programs and CCFG programs stand on the same mathematical background, and it can be shown that there exists a simple program transformation of logic programs into CCFG programs. The above topics are discussed with some examples of programs.

/

1 Introduction

Context free grammars and their variations are often regarded as the tools for representing data structures. For example, Backus-Naur Form is used for the definitions of the syntax of programming languages, the diagrams in Jackson method [1], which have the similar feature and the expressive power to those of regular expressions, subsets of context free grammars, define input/output data structures of programs, and so on[2].

Our purpose is to construct a data structure directed programming system using such context free grammars.

The first idea of our system is to regard a scheme of syntax directed translation[4] as the representations of the relation between input/output data structures. The scheme is usually interpreted as the representation of the mapping from input data to output data, but can be naturally re-interpreted as that of the relation.

The second idea is to represent an indirect relation of data structures by connecting nonterminal symbols in such schemes. A new meta symbol $\approx$ is introduced as a glue. The expressive power of schemes of the syntax directed translations is not sufficient for a practical programming system. It is shown that it becomes recursively-enumerable if we use the meta symbol $\approx$.

The formal grammar which these two ideas lead into is called Coupled Context Free Grammar(CCFG), the theorical basis of our programming system. In our system, a program is a CCFG (called a CCFG program) and programming is to define a CCFG (called CCFG programming). The denotational semantics of a CCFG program is the formal language generated by the program, a set of n-tuples of terminal strings. The intuitive meanings of a meta symbol $\approx$ are a

communication channel between programs, a conditional rule when applying a rewriting rule, and so on.

As well known, logic programs can represent the relation of objects[3] and we expect that the mathematical background of logic programs is similar to that of CCFG programs. In order to show that, we can give the simple program transformation of logic programs into CCFG programs.

In section two, CCFG and its interpretation as a programming language are informally introduced with some simple examples. In section three, the relation between logic programs and CCFG programs is described with a program transformation preserving the equivalence. In section four, the programming techniques in CCFG programs are shown with some interesting examples.

2 Basic Ideas

In this section some examples of Coupled Context Free Grammars (CCFG) are presented. These are informal introductions of CCFG programming.

2.1 Direct Relations

In syntax directed translations[4], the translation scheme for string inversion are given as follows.

Input Rules	Action Rules
$X \rightarrow \varepsilon$	$Y := \varepsilon$
$X \rightarrow aX$	$Y := Ya$
$X \rightarrow bX$	$Y := Yb$

where a and b are terminal symbols, X and Y nonterminal symbols, and ε a

null string. When an input rule is applied, the corresponding action rule in the same line must be simultaniously applied. One example of the derivations by the above scheme is given as follows when the input string is "abb":

X⇒aX	Y⇒Ya
⇒abX	⇒Yba
⇒abbX	⇒Ybba
⇒abb	⇒bba

In syntax directed translations this derivation means a translation of the input string "abb" into the output string "bba".

In CCFG programming we re-interpret the above scheme as coupling of two context free grammars and the resolvent, a couple of the derived strings ("abb","bba") as one of the denotation of the relation between the start symbols X and Y. The above scheme is rewritten as the following CCFG program:

$G_1=(N,T,S,P)$

N={X,Y}

T={a,b}

S=(X,Y)

P={{X→ε, Y→ε},
{X→aX, Y→Ya},
{X→bX, Y→Yb}},

where N is the set of nonterminal symbols, T the set of terminal symbols, S the couple of start symbols, and P the set of "rule-sets", each of which is the set of context free rules. The rule-sets are regarded as the body of the program, and the others are the declarations. The denotation D(X,Y) of the relation between two start symbols X and Y is the set of couples of terminal

strings derived by program G_1:

$$D(X,Y)=\{(\varepsilon,\varepsilon),(a,a),(b,b),(ab,ba),(ba,ab),\cdots\}$$

This is also the semantics of program G_1.

Another remarkable example of syntax directed translations is the translation of the infix notation of fomulae into the prefix notation. In CCFG programming this translation scheme is also rewritten as the following program G_2 which represents the relation between the infix notation and the prefix notation of formulae:

$$G_2=(N,T,S,P)$$

$$N=\{E_i,E_p,T_i,T_p,F_i,Fp,Id\}$$

$$T=\{a,b,c,+,*,(,)\}$$

$$S=(E_i,E_p)$$

$$P=\{\{E_i\rightarrow E_i+T_i,\ E_p\rightarrow +E_pT_p\},$$
$$\{E_i\rightarrow T_i,\quad E_p\rightarrow T_p\quad\},$$
$$\{T_i\rightarrow T_i*F_i,\ T_p\rightarrow *T_pF_p\},$$
$$\{T_i\rightarrow F_i,\quad T_p\rightarrow F_p\quad\},$$
$$\{F_i\rightarrow (E_i),\ F_p\rightarrow E_p\quad\},$$
$$\{F_i\rightarrow Id,\quad F_p\rightarrow Id\ \},$$
$$\{Id\rightarrow a\},$$
$$\{Id\rightarrow b\},$$
$$\{Id\rightarrow c\}\}.$$

For example, one derivation by program G_2 is presented as follows:

$$(E_i,\ E_p)\Rightarrow(T_i,\qquad T_p)$$
$$\Rightarrow(T_i*F_i,\qquad *T_pF_p)$$

$\Rightarrow (T_i*(E_i),\quad *T_pE_p)$

$\Rightarrow (F_i*(E_i),\quad *F_pE_p)$

$\Rightarrow (Id*(E_i),\quad *IdE_p)$

$\Rightarrow (a*(E_i),\quad *aE_p)$

$\Rightarrow (a*(E_i+T_i),\quad *a+E_pT_p)$

$\Rightarrow (a*(E_i+F_i),\quad *a+E_pF_p)$

$\Rightarrow (a*(E_i+Id),\quad *a+E_pId)$

$\Rightarrow (a*(E_i+c),\quad *a+E_pc)$

$\Rightarrow (a*(Ti+c),\quad *a+T_pc)$

$\Rightarrow (a*(F_i+c),\quad *a+F_pc)$

$\Rightarrow (a*(Id+c),\quad *a+Idc)$

$\Rightarrow (a*(b+c),\quad *a+bc).$

The resolvent ("a*(b+c)","*a+bc") is a couple of formulae in the infix notation and the prefix notation. The denotation $D(E_i,E_p)$ of the relation between the nonterminal symbols E_i and E_p is the following set:

$$D(E_i,E_p)=\{(a,a),(b,b),\cdots,(a+b,+ab),\cdots,(a*(b+c),*a+bc),\cdots\}.$$

Context free grammars is often used as the tools for representing data structures. In a similar way, context free grammars embedded in a CCFG program are also regarded as the representations of data structures in the program. For example, the following context free grammar embedded in program G_1:

$$X\rightarrow \varepsilon$$

$$X\rightarrow aX,$$

$$X\rightarrow bX,$$

is regarded as the representation of the data structure specified by the

start symbol X, and its corresponding data domain D_X is the language $\{a,b\}^*$ generated by the above context free grammar. In the same way the following context free grammar:

$$Y \rightarrow \varepsilon ,$$
$$Y \rightarrow Ya,$$
$$Y \rightarrow Yb,$$

is regarded as the data structure specified by the start symbol Y and its corresponding data domain D_Y is also $\{a,b\}^*$. The semantics of the program is represented by a subset of Cartesion product of the above two domains. In program G_2 the semantics D(X,Y) is surely the subset of $D_X \times D_Y$.

2.2.Multiple Data Structures

Next we introduce "multiple" data structures, using a new meta symbol $\approx$. This is important not only for the expressive power of CCFG to be equivalent to that of phrase structure grammars[5], but also for CCFG programming paradigms.

An example is presented as follows:

$G_3=(N,T,S,P)$

$N=\{Q,R,X,Y,Z\}$

$T=\{a,b\}$

$S=(Q,R)$

$P=\{$ $\{Q \rightarrow Z \approx X,\ R \rightarrow Y\}$ $\cdots(1)$,

$\{Z \rightarrow \varepsilon\ \}$ $\cdots(2)$, $\{X \rightarrow \varepsilon,\ Y \rightarrow \varepsilon\}$ $\cdots(4)$,

$\{Z \rightarrow abZ\}$ $\cdots(3)$, $\{X \rightarrow aX,\ Y \rightarrow Ya\}$ $\cdots(5)$,

$\{X \rightarrow bX,\ Y \rightarrow Yb\}$ $\cdots(6)\}$.

Here we find that program G_1 is embedded in program G_3 as a subprogram

(program G_1 contains rule-sets (4),(5) and (6)). A meta symbol $\approx$ appears in rule-set (1). One derivation by program G_3 is given as follows:

$(Q, R) \Rightarrow (Z \approx X, Y)$ by rule-set (1),

$\Rightarrow (abab \approx X, Y)$ by rule-sets (3),(3) and (2),

$\Rightarrow (abab \approx abab, baba)$ by rule-sets (5),(6),(5),(6) and (4).

Here because the both sides of the string "abab≈abab" are equal to each other, we say that this derivation is successful and that the couple of terminal strings ("abab", "baba") is derived by the couple of start symbols (Q,R). On the other hand, a non-successfull derivation is given as follows:

$(Q, R) \Rightarrow (Z \approx X, Y)$ by rule-set (1),

$\Rightarrow (abab \approx X, Y)$ by rule-sets (3),(3) and (2),

$\Rightarrow (abab \approx abb, bba)$ by rule-sets (5),(6),(6) and (4).

Here we say that this derivation is failed and that nothing is derived, because the both sides of the string "abab≈abb" are not equal to each other.

As described in the above two examples, a new rule for a derivation is proposed that the derivation is successful if both hand sides of each derived terminal strings which contain $\approx$ are equal to each other. Otherwise it is failed. The denotation of the relation between start symbols is the set of couples of terminal strings successfully derived by start symbols. In the case of program G_3, the denotation D(Q,R) is given as the following set:

$$D(Q,R)=\{(\alpha,\beta) \mid (Q,R) \Rightarrow (\alpha \approx \alpha, \beta)\}$$
$$=\{((ab)^n,(ba)^n) \mid n \geqq 0\}.$$

Therefore considering the data structure embedded in the above program

G_3, the data structure specified by the start symbol Q is given as the following context free grammar containing a meta symbol $\approx$:

$$Q \rightarrow Z \approx X,$$
$$Z \rightarrow \varepsilon, \quad X \rightarrow \varepsilon,$$
$$Z \rightarrow abZ, \quad X \rightarrow aX,$$
$$X \rightarrow bX.$$

Because of the above derivation rule for meta symbol $\approx$, the domain derived by the start symbol Q is obtained as the intersection of the domains derived by the symbols Z and X. Therefore we call these two data structure specified by Q "multiple".

2.3. Indirect Relations

CCFG programs represent the relations between some data structures. Particularly in the examples described above, the relations between two data structures specified by start symbols are directly represented in rule-sets. In this subsection indirect relations are proposed by the intermediation of meta symbol $\approx$.

Suppose that one program G_i represents the relation between the start symbols P and Q, and that another program G_j represents the relation between the start symbols Q and R. Then the indirect relation between symbols P and R can be represented in a new program G_k using the programs G_i and G_j as the subprograms of G_k.

For example, suppose that the following program which is an extension of G_3 is given:

$$G_4=(N,T,S,P)$$

$N=\{P,Q,R,W,X,Y,Z\}$

$T=\{a,b,c\}$

$S=(P,Q,R)$

$P=\{\quad \{P\to W,\ Q\to Z\approx X,\ R\to Y\}\ \cdots(1),$

$\{W\to\varepsilon,\ Z\to\varepsilon\}\ \cdots(2),\quad \{X\to\varepsilon,\ Y\to\varepsilon\}\ \cdots(4),$

$\{W\to cW,\ Z\to abZ\}\ \cdots(3),\quad \{X\to aX,\ Y\to Ya\}\ \cdots(5),$

$\{X\to bX,\ Y\to Yb\}\ \cdots(6)\}.$

This program represents the relation between the start symbols P, Q and R. The denotation D(P,Q,R) of program G_4 is the set of triples of terminal strings which are successfully derived by the triple of start symbols (P,Q,R) as follows:

$$D(P,Q,R)=\{(x,y,z)\mid(P,Q,R)\Rightarrow(x,y\approx y,z)\}$$
$$=\{(c^n,(ab)^n,(ba)^n)\mid n\geqq 0\}.$$

If we discard the symbol Q and think about only the relation between the start symbols P and R, the denotation of the relation can be given as the following set:

$$D(P,R)=\{(x,z)\mid(P,Q,R)\Rightarrow(x,y\approx y,z)\}$$
$$=\{(c^n,(ba)^n)\mid n\geqq 0\}.$$

This is the denotation of the indirect relation between P and R, using the intermediate symbol Q as a channel which connects the relation between P and Q with the relation between Q and R. If we have no interest in the terminal strings derived by Q, we may omit the left-hand side of the production rule $Q\to Z\approx X$ in rule-set (1). Then the program is revised as follows:

$G_5=(N,T,S,P)$

$N=\{P,R,W,X,Y,Z\}$

T={a,b,c}

S=(P,R)

P={ {P→W, Z≈X, R→Y} …(1),

{W→ε , Z→ε } …(2), {X→ε , Y→ε } …(4),

{W→cW, Z→abZ} …(3), {X→aX, Y→Ya} …(5),

{X→bX, Y→Yb} …(6)},

where rule-sets (2),…(6) are same as those in program G_4, but the second production rule in rule-set (1) is composed of only the right-hand side of the original production rule. Z≈X is interpreted as a bi-directional channel between two subprograms.

All the important concepts in CCFG programming have been introduced informally. The denotational semantics of CCFG programs can be defined with both formal languages and fixedpoint, and the operational semantics can be defined with a non-deterministic interpreter. The details of such formal discussions will be given in other papers.

3. Transformations from Logic Programs into CCFG Programs

CCFG programs have some similar properties to those of logic programs[3][5]. Both represent the relations of input/output data, both expressive powers are equivalent to that of phrase structure grammars[6], both interpreters are regarded as rewriting systems, and the executions of both interpreter are non-deterministic. Therefore it is expected that the CCFG programs are formally related with logic programs. Here we show the program transformation of logic programs to CCFG programs without proving that it preserves the equivalence.

//

Such program transformation T translates a definite Horn clause:

$$p(s_1,s_2,\cdots,s_i):\text{-}q(t_1,t_2,\cdots,t_j),\cdots,r(u_1,u_2,\cdots,u_k).,$$

into the following rule-sets:

$$\{P_1\rightarrow\bar{s}_1,\ P_2\rightarrow\bar{s}_2,\cdots,P_i\rightarrow\bar{s}_i,\ Q_1\approx\bar{t}_1,\ Q_2\approx\bar{t}_2,\cdots,Q_j\approx\bar{t}_j,\cdots,$$
$$R_1\approx\bar{u}_1,\ R_2\approx\bar{u}_2,\cdots,R_j\approx\bar{u}_k\},$$
$$\{X_1\rightarrow *\},\{X_2\rightarrow *\},\cdots,\{X_m\rightarrow *\},$$

if the definite Horn clause contains variables $X_1,X_2,\cdots,X_m$. Here * is anonymous nonterminal symbol which can derive any terminal string, i.e. $*\Rightarrow T^*$. The translated strings $\bar{s}_1,\cdots,\bar{u}_k$ are the strings corresponding to the terms $s_1,\cdots,u_k$, substituting nonterminal symbols into the corresponding variables in the terms and terminal symbols into the corresponding constants and functors.

Some typical examples of program translations are discussed as follows.

First of all we consider the definite Horn clause in which a variable appears once in the head and more than once in the body as follows:

$$p(X):\text{-}q(X),r(X).$$

Here the variable X is shared with the predicate symbols p,q and r. This is translated into

$$\{P_1\rightarrow X,\ Q_1\approx X,\ R_1\approx X\},$$
$$\{X\rightarrow *\}.$$

Eliminating the variable X, the above rule-sets can be easily transformed into

$$\{P_1\rightarrow Q_1\approx R_1\}.$$

We find that the nonterminal symbol P_1 has a multiple data structure.

In this way, if a variable appears once in the head of an original definite clause and more than once in the body, the translated rule-set contains a multiple data structure.

Next we consider the following definite clause:

$$p(X):-q(X,Y),r(Y).,$$

which contains the internal variable Y in its body. This is translated into

$$\{P_1 \rightarrow X,\ Q_1 \approx X,\ Q_2 \approx Y,\ R_1 \approx Y\},$$
$$\{X \rightarrow *\},\{Y \rightarrow *\}.$$

Eliminating the variables X and Y , the above rule-sets are easily translated into

$$\{P_1 \rightarrow Q_1,\ Q_2 \approx R_1\}.$$

In the original clause, the internal variable Y indicates that the second argument of predicate q and the first argument of predicate r are resolved as same ground terms. In the translated rule-set, the production rule $Q_2 \approx R_1$ also indicates that nonterminal symbols Q_2 and R_1 are resolved as same terminal strings.

In this way, if there exist internal variables in an original definite clause, there exist production rules which contains meta symbol $\approx$ and does not have its left-hand side nonterminal symbol in the translated rule-set.

As described above, we find that the interpretations of logic programs are similar to that of CCFG programs. Using above transformation, we may find new relations between some concepts in logic programs and in CCFG programs.

4.Programming Examples

In this section some programming techniques are discussed with some examples. First in subsection 4.1 the programs to parse and translate strings are given. In subsection 4.2 the program to process streams of integers is given.

4.1.Parsing Strings

The first application is to generate a parser just like a syntax directed translator. Because the origin of CCFG is a syntax directed translator, we can straightforwardly construct it in CCFG programs.

One example of such programs is program G_2 in section 2.1, which represents the relation between the formula E_i in infix notation and the formula E_p in prefix notation. Using this program, the program to parse the input string "a*(b+c)" in infix notation and translate it into the output string "*a+bc" in prefix notation, is given as follows:

PARSE=(N',T,S',P')

N'={Out}∪N,

S'=Out,

P'={Main'}∪P,

Main'={a*(b+c)≈E_i, Out→E_p},

where N, T and P have been already defined in program G_2 in section 2.1. Program PARSE can obtain the formula "*a+bc" in prefix notation corresponding to the formula "a*(b+c)" in infix notation. One successful derivation by this program is as follows:

$$\text{Out} \Rightarrow (E_p,\ a*(b+c) \approx E_i)$$
$$\overset{*}{\Rightarrow} (*a+bc,\ a*(b+c) \approx a*(b+c)).$$

Here the derivation in the second line is same as the derivation in subsection 2.1. Since the string "$a*(b+c)\approx E_i$" means that E_i must derive the terminal string "$a*(b+c)$", the string represents the parsing of "$a*(b+c)$" by E_i.

Another example is to concurrently parse both imcomplete formulae in infix and prefix notations and to obtain the complete ones. We suppose that the formula in infix notation is given as "$a*\square\square+c)$" and that the formula in prefix notation is given as "$*\square\square b\square$", where each $\square$ means one terminal symbol. The program to obtain the complete formulae is given as follows:

PARSE2=(N",T,S",P")

N"=$\{S_i, S_p, \square_1, \square_2, \cdots, \square_5\} \cup$ N,

S"=$\{S_i, S_p\}$

P"={Main"}$\cup$P$\cup\square$

Main"=$\{S_i \rightarrow a*\square_1\square_2+c) \approx E_i,\ S_p \rightarrow *\square_3\square_4 b\square_5 \approx E_p\}$,

$\square = \{\{\square_1 \rightarrow a\}, \{\square_1 \rightarrow b\}, \{\square_1 \rightarrow c\}, \{\square_1 \rightarrow +\}, \{\square_1 \rightarrow *\},$
$\{\square_1 \rightarrow (\}, \{\square_1 \rightarrow)\}, \cdots, \{\square_5 \rightarrow a\}, \{\square_5 \rightarrow b\}, \{\square_5 \rightarrow c\},$
$\{\square_5 \rightarrow +\}, \{\square_5 \rightarrow *\}\}$,

where N, T and P have been already defined in program G_2 in section 2.1. Nonterminal symbols $\square_i$ ($i=1,\cdots,5$) are used in place of $\square$s because each $\square_i$ may derive a terminal string different from each other. Program PARSE2 has one solution and a successful derivation is given as follows:

$$(S_i, S_p) \Rightarrow (a*\square_1\square_2+c) \approx E_i,\ *\square_3\square_4 b\square_5 \approx E_p)$$
$$\overset{*}{\Rightarrow} (a*\square_1\square_2+c) \approx a*(b+c),\ *\square_3\square_4 b\square_5 \approx *a+bc)$$

$\overset{*}{\Rightarrow}$(a*(b+c)≈a*(b+c), *a+bc≈ *a+bc).

Though effective strategies to derive the solution are not discussed in this paper, we can effectively compute the above derivation by using the LL(k)(Left to right Leftmost derivation) or RR(k)(Right to left Rightmost derivation) parsing techniques.

As described above, a string $x \approx X$ represents the parsing of a terminal string x by X. Extending the concept of such parsing, we can assume that a string $\alpha \approx \beta$, where α and β are strings of terminal symbols and nonterminal symbols, also represents an extended parsing. Therofore the implementations of a CCFG intepreter is to develop such extended parsers to effectively compute above derivations.

4.2.Processing Streams

Stream is an important data structure to naturally express data flows. Here we simulate the streams and their processings in CCFG programming system.

Let us consider the calculation to obtain the sum of squared integer streams. For example, if the input stream is <1,2>, the corresponding calculation result is five, because $1^2+2^2 = 5$. This computation is designed as follows:

<1,2> ⇒ [SQR] ⇒ <1,4> ⇒ [SUM] ⇒ 5,

where the program SQR consumes a stream of integers and produces the stream of the squared integers, and the program SUM consumes it and calculates the sum of all its elements.

The CCFG program which executes the above computation is given. First the declaration parts and main program are as follows:

SUMSQR=(N,T,S,P)

N={Result, SqrStI, SqrStO, SumStI, SumO}

T={☆, |}

S=Result

P={Main} U Number U SQR U SUM U MUL

Main={☆|☆☆|≈SqrStI, SqrStO≈SumStI, Result→SumO},

where the output nonterminal symbol "Result" derives the computation result, the nonterminal symbols SqrStI and SqrStO are the input/output nonterminal symbols of subprogram SQR and the nonterminal symbols SumStI and SumO are the input/output nonterminal symbols of subprogram SUM. Since in CCFG programming system the primitive objects which represent integers are not prepared, integers are expressed with the strings of the terminal symbol ☆, which value as an integer is its length. Namely the data structure of integers are the following rule-sets:

Number={{Num→ ε },{Num→☆Num}}.

A stream of integers are expressed as a string of integers in the following rule-sets:

{NumSt→ ε },{NumSt→Num|NumSt},

where symbol "|" is a delimiter between integers. So the string "☆|☆☆|" means a stream of integer <1,2>.

The subprogram SQR is the following rule-sets:

SQR={{SqrStI→ ε , SqrStO→ ε },
{SqrStI→SqrI|SqrStI, SqrStO→SqrO|SqrStO},
{ SqrI→M1≈M2, SqrO→MR}},

where the rule-set in the first line means that if the input stream SqrStI

of SQR is empty, then the output stream SqrSt0 of SQR is also empty. The rule-set in the second line means that if the input stream SqrStI is the concatenation of a integer SqrI and its following input stream, then the output stream SqrSt0 is the concatenation of integer Sqr0 and its following output stream. The rule-set in the third line means that Sqr0 is the result MR of a multiplier MUL where its two input arguments M1 and M2 are both equal to SqrI.

The multiplier is given as the following rule- sets:

$$\begin{aligned} MUL=\{&\{M1\rightarrow \varepsilon,\quad M2\rightarrow Num,\ MR\rightarrow \varepsilon\},\\ &\{M1\rightarrow \bigstar M1,\ M2\rightarrow M2,\quad MR\rightarrow M2\ MR\}\}. \end{aligned}$$

The rule-set in the first line means that the result MR is zero where the input argument M1 is zero. The rule-set in the second line means that if the input argument M1 is 1+M1 then the result MR is M2+MR for any multiplicand M2. This is the recursive definition of triples (M1,M2,MR). Because an integer is expressed by the number of ☆s, the concatenation of two integers M2 and MR means the sum M2+MR.

The subprogram SUM is given as the following rule-sets:

$$\begin{aligned} SUM=\{&\{SumStI\rightarrow \varepsilon,\qquad Sum0\rightarrow \varepsilon\},\\ &\{SumStI\rightarrow Num|SumStI,\ Sum0\rightarrow Num\ Sum0\}\}, \end{aligned}$$

where the concatenation of two integers Num and Sum0 in the second line means the sum Num+Sum0.

One example of computation by this program is the following successfull derivations:

$$Result\Rightarrow(\bigstar|\bigstar\bigstar|\approx SqrStI,\qquad SqrSt0\approx SumStI,\qquad Sum0)$$

⇒(☆|☆☆|≈SqrI|SqrStI, SqrO|SqrStO≈SumStI, SumO)
⇒(☆|☆☆|≈M1|SqrStI≈M2|SqrStI, MR|SqrStO≈SumStI, SumO)
⇒(☆|☆☆|≈☆M1|SqrStI≈M2|SqrStI, M2MR|SqrStO≈SumStI, SumO)
⇒(☆|☆☆|≈☆|SqrStI≈Num|SqrStI, Num|SqrStO≈SumStI, SumO)
≒(☆|☆☆|≈☆|SqrStI≈☆|SqrStI, ☆|SqrStO≈SumStI, SumO)

The tuple of sentential forms in the last line is equivalent to the following one:

(☆|☆☆|≈☆|SqrStI, ☆|SqrStO≈SumStI, SumO).

Then we can use this tuple and go on with the calculation:

≒(☆|☆☆|≈☆|☆☆|, ☆|☆☆☆☆|≈SumStI, SumO)
≒(☆|☆☆|≈☆|☆☆|, ☆|☆☆☆☆|≈☆|SumStI, ☆SumO)
≒(☆|☆☆|≈☆|☆☆|, ☆|☆☆☆☆|≈☆|☆☆☆☆|, ☆☆☆☆☆).

At last, the start symbol Result has derived interger five, i.e. Result≒☆☆☆☆☆.

In this way, processing streams can be simulated in CCFG programming system. We first made subprograms to process streams, and connected them by the intermediation of meta symbols ≈.

5.Discussion

In this paper Coupled Context Free Grammars and its interpretation as a programming languages are informally introduced. Here the strict definitions on them are not given. Such definitions and formal discussions will be described in other papers.

The following problems remain.

(1) to devise effective parsing methods for CCFG.

(2) to design a practical programming language based on CCFG and implement such language.

(3) to compare CCFG programs with logic programs and clarify the properties of CCFG.

References

[1]Jackson,M.A.:"Principles of Program Design", Academic Press(1975).

[2]Abramson,H.:Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars, Proceedings of the International Conference on Fifth Generation Computer System(1984).

[3]Kowalski,R.:Predicate Logic as Programming Language, Information Processing 74 (1974).

[4]Aho,A.V. and Ullman,J.D.:"The Theory of Parsing, Translation and Compiling Vol1:Parsing", Prentice Hall(1972).

[5]Yamashita,Y. and Nakata,I.:The Extenstion of Context Free Grammars and the Programming Language Based on Coupled Grammars, IPSJ Working Group of Software Foundation 15-5(1985), in Japanese.

[6]Tarnlund,S-A.:Horn Clause Computability, BIT 17(1977),pp.215

[7]Nakata,I. and Yamashita,Y.:Program Transformation in Coupled Context Free Grammars, IPSJ Working Group of Software Foundation 17-3(1986), in Japanese.